FIT3088 Computer Graphics
Semester 2,

Assignment 2: 3D Globe and Satellite Visualisation with WebGL

Version 1.0, 22 September 2018

Important Information

- Due date: Friday 26 October 2018 at 11:55PM (end of Week 13).
- Total marks: 20. This assignment counts towards 20% of your final mark for this unit.
- Late penalty: Minus 10% per day.
- Platform: You need to use WebGL for this assignment. Your program needs to run with the current version of Firefox or Chrome.

Submission

You need to submit your program for assessment via Moodle. The submission should be a zipped file with the name A2-StudentId.zip where StudentId is your Monash Student ID number (e.g. an ID of 12345678 would result in A1-12345678.zip).

The zip file should include:

- All source code: an index.html file, JavaScript files, and other resource files (raster images for textures, OBJ file, etc.).
- A readme file describing your code structure, if necessary instructions on how to use your program, and a description of any interesting or supplementary features we should look out for.

Assignment description

The outcome of this assignment is an interactive globe with the current position of GPS satellites. The user can rotate the globe and click on a satellite to display the satellite name. The globe is textured and shows the current daylight area.

The goal is to apply interactive three-dimensional transformations and coordinate systems, 3D rendering, shading, texture mapping and shader programming.

Program Specification

- The final display will contain:
    - A dark grey or black background.
    - A globe at the centre of the canvas, textured with two blended images showing night lights (for areas currently not in daylight) and a land-cover

texture (for areas currently in daylight). The two textures are blended based on the current sun position relative to Earth.
    - 3D satellites representing each of the currently existing GPS satellites at their current real position. The satellites are scaled up relative to their real size, such that they appear as small models on the display. The satellites face the globe always with the same face.
    - The orbit of each GPS satellite is represented as a thin white line around the globe.
- Perspective camera, pointing at the centre of the globe, with a field of view that shows satellites close to the camera larger than satellites in the background, but does not overly distort the geometry of the globe.
- Diffuse shading for the globe and the satellites with a point light source placed on the top right relative to the camera view direction.
- The globe together with the satellites and the orbits are rotatable (either with sliders or by mouse interaction).
- The user can click on each satellite, which displays the name of the selected satellite on the HTML page.
- The satellites can optionally be animated, such the satellite rotate around the globe once within a few seconds. The animation can be toggled on and off by the user.

Hints

Multiple models and shader programs: It is highly recommendable that you develop individual shader programs for (1) the globe, (2) the satellites and (3) the lines for the satellite orbits. Hence you will manage three different geometry models, and will create a shader program for each model. In the render loop, you will switch between the three programs. It is important to structure your JavaScript code accordingly. See here for how to work with multiple models and shader programs:
https://webglfundamentals.org/webgl/lessons/webgl-drawing-multiple-things.html

Globe geometry: There are different ways how a sphere can be approximated by triangles. A simple method is by lines of longitude and latitude (see the textbook, section 2.4.3, 'Approximating a Sphere', p. 87). It is recommended that you start with this method. However, this method creates very thin triangles near the poles that can lead to lighting and texturing artefacts. If you are looking for an extra challenge and bonus points, you can use a better method that approximates the sphere by recursive subdivision (see the textbook, section 6.6 'Approximation of a Sphere by Recursive Subdivision', p. 327). Note: The radius of Earth is 6371 km.

Globe normal vectors: For a sphere, there is no need to compute normal vectors with JavaScript and then load the normals into a buffer on the GPU. The vertex shader can instead simply derive the normals from the vertex position.

Globe texture: Raster images for the day and the night textures are provided on Moodle. To blend the two textures, create a third greyscale texture image, where white

indicates a sunlit area and black a night area. When applying the textures in the fragment shader, use the greyscale texture to blend between the day and night textures. Use suncalc.js at https://github.com/mourner/suncalc to create a greyscale image for the current time. To do so, crate a two-dimensional array for regular steps in longitude and latitude. Then use

```
var times = SunCalc.getTimes(now, latitudeDeg, longitdueDeg);
```

to compute the time of sunrise and sunset for the current day, and compare sunrise and sunset times to the current time. The current time is returned with the following call:

```
var now = new Date();
```

You can compare the current time to the sunrise and sunset times to determine a black or white value:

```
var value = now < times.sunrise || now > times.sunset ? 0 : 255;
```

On 22 September 2018, 6:10PM, this resulted in the following image (for a 360×180 grid). The central meridian at 0 degrees longitude of London is at the centre of the image. The black area on the right side shows that night was just about to start in Melbourne.

`SunCalc.getTimes()` also returns dusk and dawn times. Use these values to compute a continuous gradient between white and black.

Satellite model: Load a generic satellite model from an external file. It is suggested you use the OBJ file format. A model is provided on Moodle but you can use other satellite models. Only load the satellite geometry once onto the GPU but render one instance for each satellite.

Satellite positions: According to Wikipedia, there are currently 32 GPS satellites (31 in use). Satellite positions are commonly encoded in the TLE (Two-Line Element) text format, which the satellite.js library at https://github.com/shashwatak/satellite-js can interpret. The current GPS satellite orbits in TLE format are available here: http://www.celestrak.com/NORAD/elements/gps-ops.txt. Download this file and store it with your other project files. The TLE information is valid for quite a bit into the future, so we can use it for our real-time application. In your JavaScript code, load this file with an asynchronous XMLHTTPRequest and parse it with satellite.js to get the position of a satellite for a given time. There is an example project on Moodle with code that does all this and contains additional explanations.

Satellite orbits: GPS Satellites circulate around Earth on elliptic orbits. The orbital radius is approximately 26,600 km. See the example project on Moodle for explanations on how you can extract the geometry of an orbit. Render the orbit for each satellite with a line.

Expected outcome and marking criteria

Functionality: 60% total, 15% for each of the following four types of functionality:

1. **Modelling and 3D transformations**: Satellites are placed at correct distance relative to the diameter of Earth. The user can rotate the globe, and the satellites and orbits are rotated with the globe. Satellites always face the centre of the globe. The camera is setup correctly, looking at the centre of the globe.
2. **Lighting and shading**: Diffuse shading and ambient shading is applied to the globe and the satellites. The point light source does not rotate with the globe.
3. **Texturing**: Day and night textures are correctly blended for the current time.
4. **Picking and Animation**: Satellites can be clicked and the name about the selected satellite is displayed on the HTML page. The satellite animation can be switched on and off and the satellite positions are correct during the animation.

Rendering efficiency and graphical quality: 10%
We are looking for visually pleasing high-quality rendering: Reduce aliasing in textures using mipmaps and anisotropic texture filtering (if available).
We are also looking for efficient code: Enable backface culling; compute normals for the globe in the vertex shader. Make sure a single copy of the satellite geometry is loaded onto the GPU, and all satellite instances use this same geometry.

JavaScript programming and code structure: 20%
We expect you to create a logical object-oriented code structure. We recommend you place your JavaScript code in one or multiple external JS files. You can add your GLSL code to the HTML file (as we did in previous tutorials) or you can place your GLSL code in external files and import those files. You may not use third-party libraries with the following exceptions:

- Libraries that we used in any of the lab tutorials, that is, MV.js, initShaders.js, and webgl-utils.js.
- You can use any library for loading Wavefront OBJ files.
- You can use any library for reading shader code from external files (if you prefer storing your GLSL code in external files).
- suncalc.js at https://github.com/mourner/suncalc for computing sun positions.
- satellite.js at https://github.com/shashwatak/satellite-js for computing satellite positions.

Code readability, formatting and documentation: 10%
We expect you to write readable code that complies with common coding rules for JavaScript and HTML. We expect you to use strict mode for JavaScript: place
`"use strict";`
at the beginning of each script file. For more information about strict mode, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Your code should pass a test with JSLint, ESLint, or a similar JavaScript syntax checker. Rules in syntax checkers vary with the personal preferences of their authors. We will

therefore not be super strict on this requirement, but you need to use common sense and make sure your JavaScript code and HTML DOM are in good shape. This includes:

- Properly indented code and usage of empty lines to structure your code. Use an IDE with proper automatic formatting and indenting for JavaScript and HTML.
- Apply common coding conventions for naming functions, variables and constants.
- Add documentation for all non-trivial functions, variables and code segments.
- In vertex and fragment shader code, start attribute names with a_, uniform names with u_ and varying names with v_. The underscores are optional.

========== End of the document ==========